A Unified Peer-to-Peer Database Framework
and its Application for Scalable Service Discovery

Wolfgang Hoschek
CERN IT Division
European Organization for Nuclear Research
1211 Geneva 23, Switzerland
Wolfgang.Hoschek@cern.ch

Abstract

In a large distributed system spanning many administrative domains such as a DataGrid, it is often desirable to maintain and query dynamic and timely information about active participants such as services, resources and user communities. However, in such a database system, the set of information tuples in the universe is partitioned over one or more distributed nodes, for reasons including autonomy, scalability, availability, performance and security. It is not obvious how to enable powerful discovery query support and collective collaborative functionality that operate on the distributed system as a whole, rather than on a given part of it. Further, it is not obvious how to allow for search results that are fresh, allowing dynamic content. It appears that a Peer-to-Peer (P2P) database network may be well suited to support dynamic distributed database search, for example for service discovery. In this paper, we take the first steps towards unifying the fields of database management systems and P2P computing, which so far have received considerable, but separate, attention. We extend database concepts and practice to cover P2P search. Similarly, we extend P2P concepts and practice to support powerful general-purpose query languages such as XQuery and SQL. As a result, we devise the *Unified Peer-to-Peer Database Framework (UPDF)*, which is unified in the sense that it allows to express specific applications for a wide range of data types, node topologies, query languages, query response modes, neighbor selection policies, pipelining characteristics, timeout and other scope options.

1 Introduction

The next generation Large Hadron Collider (LHC) project at CERN, the European Organization for Nuclear Research, involves thousands of researchers and hundreds of institutions spread around the globe. A massive set of computing resources is necessary to support it's data-intensive physics analysis applications, including thousands of network services, tens of thousands of CPUs, WAN Gigabit networking as well as Petabytes of disk and tape storage [1]. To make collaboration viable, it was decided to share in a global joint effort - the European DataGrid (EDG) [2, 3, 4, 5] - the data and locally available resources of all participating laboratories and university departments.

Grid technology attempts to support flexible, secure, coordinated information sharing among dynamic collections of individuals, institutions and resources. This includes data sharing but also includes access to computers, software and devices required by computation and data-rich collaborative problem solving [6]. These and other advances of distributed

computing are necessary to increasingly make it possible to join loosely coupled people and resources from multiple organizations.

An enabling step towards increased Grid software execution flexibility is the (still immature and hence often hyped) *web services* vision [2, 7, 8] of distributed computing where programs are no longer configured with static information. Rather, the promise is that programs are made more flexible and powerful by querying Internet databases (registries) at runtime in order to discover information and network attached third-party building blocks. Services can advertise themselves and related metadata via such databases, enabling the assembly of distributed higher-level components. For example, a data-intensive High Energy Physics analysis application sweeping over Terabytes of data looks for remote services that exhibit a suitable combination of characteristics, including network load, available disk quota, access rights, and perhaps Quality of Service and monetary cost.

More generally, in a distributed system, it is often desirable to maintain and query dynamic and timely information about active participants such as services, resources and user communities. As in a data integration system [9, 10, 11], the goal is to exploit several independent information sources as if they were a single source. However, in a large distributed database system spanning many administrative domains, the set of information tuples in the universe is partitioned over one or more distributed nodes, for reasons including autonomy, scalability, availability, performance and security. It is not obvious how to enable powerful discovery query support and collective collaborative functionality that operate on the distributed system as a whole, rather than on a given part of it. Further, it is not obvious how to allow for search results that are fresh, allowing time-sensitive dynamic content. It appears that a Peer-to-Peer (P2P) database network may be well suited to support dynamic distributed database search, for example for service discovery.

The overall P2P idea is as follows. Rather than have a centralized database, a distributed framework is used where there exist one or more autonomous database nodes, each maintaining its own data. Queries are no longer posed to a central database; instead, they are recursively propagated over the network to some or all database nodes, and results are collected and send back to the client. The key problems then are:

- *What are the detailed architecture and design options for P2P database searching? What response models can be used to return matching query results? How should a P2P query processor be organized? What query types can be answered (efficiently) by a P2P network? What query types have the potential to immediately start piping in (early) results? How can a maximum of results be delivered reliably within the time frame desired by a user, even if a query type does not support pipelining? How can loops be detected reliably using timeouts? How can a query scope be used to exploit topology characteristics in answering a query?*

- *Can we devise a unified P2P database framework for general-purpose query support in large heterogeneous distributed systems spanning many administrative domains? More precisely, can we devise a framework that is unified in the sense that it allows to express specific applications for a wide range of data types, node topologies, query languages, query response modes, neighbor selection policies, pipelining characteristics, timeout and other scope options?*

In this paper, we take the first steps towards unifying the fields of database management systems and P2P computing, which so far have received considerable, but separate, attention. We extend database concepts and practice to cover P2P search. Similarly, we extend P2P

concepts and practice to support powerful general-purpose query languages such as XQuery [12] and SQL [13]. As a result, we answer the above questions by proposing the so-called *Unified Peer-to-Peer Database Framework (UPDF)*.

This paper is organized as follows. Section 2 introduces a query, data and database model. The related but orthogonal concepts of (logical) *link topology* and (physical) *node deployment model* are discussed. Definitions are proposed, clarifying the notion of *node*, *service*, *fat*, *thin* and *ultra-thin* P2P networks, as well as the commonality of a P2P network and a P2P network for service discovery. The *agent P2P model* is proposed and compared with the *servent P2P model*. A timeout-based mechanism to reliably detect and prevent query loops is proposed.

Section 3 characterizes in detail four techniques to return matching query results to an originator, namely *Routed Response*, *Direct Response*, *Routed Metadata Response*, and *Direct Metadata Response*. We discuss to what extent a given P2P network must mandate the use of any particular response mode throughout the system.

Section 4 unifies query processing in centralized, distributed and P2P databases. A theory of query processing for queries that are (or are not) *recursively partitionable* is proposed, which directly reflects the basis of the P2P scalability potential. The definition and properties of *simple*, *medium* and *complex* queries are clarified with respect to recursive partitioning. It is established to what extent simple, medium and complex queries support pipelining.

Section 5 proposes *dynamic abort timeouts* using as policy *exponential decay with halving*. This ensures that a maximum of results can be delivered reliably within the time frame desired by a user even if a query does not support pipelining. It is established that a loop timeout must be static.

Section 6 uses the concept of a *query scope* to navigate and prune the link topology and filter on attributes of the deployment model. Indirect specification of scope based on neighbor selection, timeout and radius is detailed.

Section 7 compares our work with existing research results. Finally, Section 8 concludes this paper. We also outline interesting directions for future research.

2 Background

2.1 Database Model and Topology

A node holds a set of tuples in its database. A database may be anything that accepts queries from the query model and returns results according to the data model (see below). For example, in [2, 14] we have introduced a registry node for service discovery that maintains hyperlinks and cache content pointed to by these links. A content provider can publish a hyperlink, which in turn enables the registry (and third parties) to pull (retrieve) the current content. A remote client can query a registry in the XQuery language, obtaining a set of tuples as answer. Figure 1 illustrates such a registry node with several content providers and clients.

A distributed database framework is used where there exist one or more nodes. Each node can operate autonomously. A node holds a set of tuples in its database. A given database belongs to a single node. For flexibility, the databases of nodes may be deployed in any arbitrary way (*deployment model*). For example, a number of nodes may reside on the same host. A node's database may be co-located with the node. However, the databases of all nodes may just as well be stored next to each other on a single central data server.

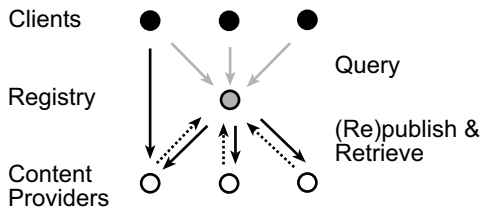

Figure 1: Clients, Registry and Content Providers [2].

The set of tuples in the universe is partitioned over the nodes, for reasons including autonomy, scalability, availability, performance and security. Nodes are interconnected with links in any arbitrary way. A link enables a node to query another node. A *link topology* describes the link structure among nodes. The centralized model has a single node only. For example, in a service discovery system, a link topology could tie together a distributed set of administrative domains, each hosting a registry node holding descriptions of services local to the domain. Several link topology models covering the spectrum from centralized models to fine-grained fully distributed models can be envisaged, among them single node, star, ring, tree, semi hierarchical as well as graph models. Real-world distributed systems often have a more complex organization than any simple topology. They often combine several topologies into a hybrid topology. Nodes typically play multiple roles in such a system. For example, a node might have a centralized interaction with one part of the system, while being part of a hierarchy in another part [15]. Figure 2 depicts some example topologies.

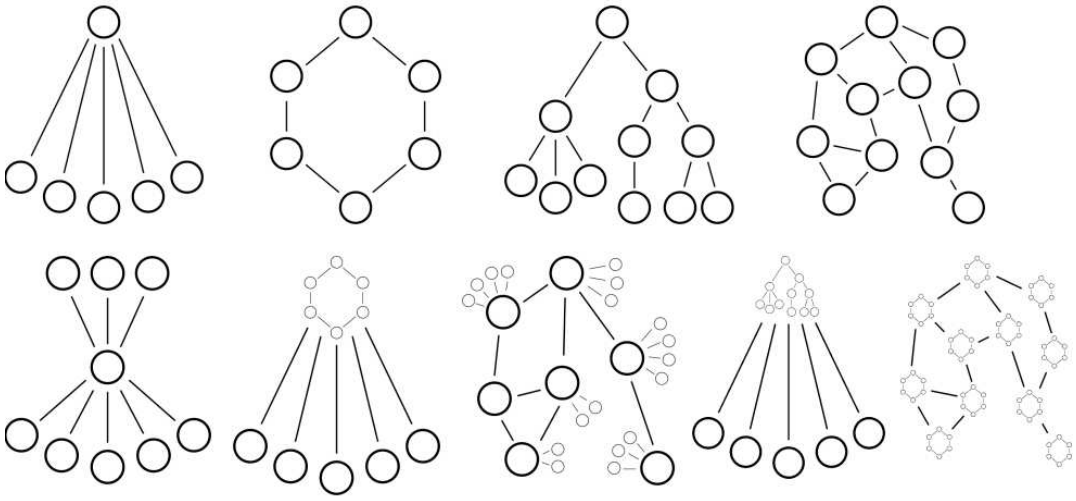

Figure 2: Example Link Topologies [15].

Clearly not all nodes in a topology are equal. For example, node bandwidth may vary by four orders of magnitude (50Kbps-1000Mbps), latency by six orders of magnitude (10us-10s), and availability by four orders of magnitude (1%-99.99%).

We stress that it is by no means justifiable to advocate the use of graph topologies irrespective of application context and requirements. Depending on the context, all topologies have their merits and drawbacks in terms of scalability, reliability, availability, content coherence, fault tolerance, security and maintainability. However, from the structural perspective, the graph topology is the most general one, being able to express all other conceivable topolo-

gies. Since our goal is to support queries that are generally independent of the underlying topology, this paper discusses problems arising in graph topologies. A problem solution that applies to a graph also applies to any other topology. Of course, a simpler or more efficient solution may exist for any particular topology. The results of this paper help *enable* the use of graph topologies where appropriate, they do not *require* or *mandate* the use of them.

2.2 Query and Data Model

We have introduced a dynamic data model for discovery in [2, 14]. It is a general-purpose data model that operates on *tuples.* Briefly, a tuple is an annotated multi-purpose soft state data container that may contain a piece of arbitrary *content* and allows for refresh of that content at any time. Content can be structured or semi-structured data in the form of any arbitrary well-formed XML [16] document or fragment. An individual tuple may, but need not, have a schema (XML Schema [17]), in which case it must be valid according to the schema. All tuples may, but need not, share a common schema. This flexibility is important for integration of heterogeneous content. Examples for content include a service description expressed in WSDL [18], a file, picture, current network load, host information, stock quotes, etc. Discussion in this paper often uses examples where the term *tuple* is substituted by the more concrete term *service description.* Consider the following example tuple set:

```
<tupleset>
    <tuple link="http://registry.cern.ch/getServiceDescription"
           type="service" ctx="parent" TS1="10" TC="15" TS2="20" TS3="30">
       <content>
         <service>
           <interface type ="http://gridforum.org/interface/Presenter-1.0">
             <operation>
               <name>XML getServiceDescription()</name>
               <bind:http verb="GET" URL="https://registry.cern.ch/getServiceDescription"/>
             </operation>
           </interface>

           <interface type = "http://gridforum.org/interface/XQuery-1.0">
             <operation>
               <name> XML query(XQuery query)</name>
               <bind:beep URL="beep://registry.cern.ch:9000"/>
             </operation>
           </interface>
         </service>
       </content>
    </tuple>

    <tuple link="http://repcat.cern.ch/getServiceDescription?id=4711"
           type="service" ctx="child" TS1="30" TC="0" TS2="40" TS3="50">
    </tuple>
</tupleset>
```

Our general-purpose query model is intended for read-only search. Insert, update and delete capabilities are not required and not addressed. We have defined these capabilities elsewhere [2, 14]. A *query* is formulated against a global database view and is insensitive to link topology and deployment model. In other words, to a query the set of all tuples appears as a single homogenous database, even though the set may be (recursively) partitioned across many nodes and databases. This means that in a relational or XML environment, at the global level, the set of all tuples appears as a single, very large, table or XML document,

respectively. The *query scope*, on the other hand, is used to navigate and prune the link topology and filter on attributes of the deployment model. Searching is primarily guided by the query. Scope hints are used only as necessary. A query is evaluated against a set of tuples. The set, in turn, is specified by the scope. Conceptually, the scope is the input fed to the query. The query scope is a set and may contain anything from all tuples in the universe to none. Consider the example discovery queries:

- *Simple Query: Find all (available) services.*
- *Simple Query: Find all services that implement a replica catalog service interface and that CMS members are allowed to use, and that have an HTTP binding for the replica catalog operation "XML getPFNs(String LFN)".*
- *Simple Query: Find all CMS replica catalog services and return their physical file names (PFNs) for a given logical file name (LFN); suppress PFNs not starting with "ftp://".*
- *Medium Query: Return the number of replica catalog services.*
- *Complex Query: Find all (execution service, storage service) pairs where both services of a pair live within the same domain. (Job wants to read and write locally).*

For a detailed discussion of a wide range of discovery queries, their representation in the XQuery language, as well as detailed motivation and justification, see our prior studies [2].

2.3 Definitions - Service and Node

Let us clarify the notion of node and service. A *service* exposes some functionality in the form of service interfaces to remote clients. Example services are an echo service, a job scheduler, a replica catalog, a time service, a gene sequencing service and a language translation service. A *node* is a service that exposes *at least* functionality (i.e. service interfaces) for publication and P2P queries. Examples are a *hyper registry* as introduced in our prior studies [2, 14], a Gnutella [19] file sharing node and an extended job scheduler. Put another way, any service that happens to support publication and P2P query interfaces is a node. This implies that *every node is a service*. It does not imply that every service is a node. Only nodes are part of the P2P topology, while services are not, because they do not support the required interfaces. Usually, most services are not nodes. However, in some networks most or all services are nodes.

- **Most services are not nodes.** We propose to speak of a *fat* P2P network. Typically, only one or a few large and powerful services are nodes, enabling publication and P2P queries. An example is a backbone network of 10 large registry nodes that ties together 10 administrative domains, each hosting a registry node to which local domain services can publish to be discovered, as depicted in Figure 3 (left). The services (shown small on the edges) are not part of the network.
- **Most or all services are nodes.** We propose to speak of a *thin* or *ultra-thin* P2P network. An example is a network of millions of small services, each having some proprietary core functionality (e.g. replica management optimization, gene sequencing, multi-lingual translation), actively using the network for searching (e.g. to discover replica catalogs, remote gene mappers or language dictionary services), but also actively contributing to its search capabilities, as depicted in Figure 3 (right).

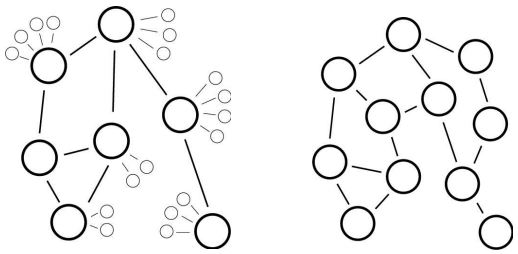

Figure 3: Fat (left) and Ultra-thin (right) Peer-to-Peer network.

There is no difference between these scenarios in terms of technology. Discussion in this paper is applicable to ultra-thin, thin and fat P2P networks. For simplicity of exposition, examples illustrate ultra-thin networks (every service is a node).

2.4 P2P network vs. P2P network for service discovery

In any kind of P2P network, nodes may publish themselves to other nodes, thereby forming a topology. In a P2P network for service discovery, services and other content providers may publish their service link and content links to nodes. Because nodes are services, also nodes may publish their service link (and content links) to other nodes, thereby forming a topology. In any kind of P2P network, a node has a database or some kind of data source against which queries are applied. In a P2P network for service discovery, this database happens to be the publication database. In other words, publication enables topology construction and at the same time constructs the database to be searched. Discussion in this paper is applicable to any kind of P2P network, while the examples illustrate service discovery.

2.5 Agent P2P Model

Queries in what we propose as the *agent P2P model* flow as follows. When any *originator* wishes to search the P2P network with some query, it sends the query to a single node. We call this entry point the *agent* node of the originator (i.e. its service gateway). The agent applies the query to its local database and returns matching results; it also forwards the query to its neighbor nodes. These neighbors return their local query results; they also forward the query to their neighbors, and so on.

For flexibilty, the protocol between originator and agent is left unspecified. The agent P2P model is a hybrid of centralization and decentralization. It allows fully decentralized infrastructures, yet also allows seamless integration of centralized client-server computing into an otherwise decentralized infrastructure. An originator may embed its agent in the same process (decentralized). However, the originator may just as well choose a remote node as agent (centralized), for reasons including central control, reliability, continuous availability, maintainability, security, accounting and firewall restrictions on incoming connections for originator hosts. For example, a simple HTML GUI may be sufficient to originate queries that are sent to an organization's agent node. Note that only nodes are part of the P2P topology, while the originator is not, because it does not possess the functionality of a node. The agent P2P model provides location and distribution transparency to originators. An originator is unaware that (and how) database tuples are partitioned among nodes. It only communicates with an agent black box.

2.6 Servent P2P Model

In contrast, in the *servent P2P model* (e.g. Gnutella) there exists no agent concept, but only the concept of a servent. Put another way, the agent is always embedded into the originator process, forming a monolithic servent. This model is decentralized, and it does not allow for some degree of centralization. This restriction appears unjustified. More importantly, it seriously limits the applicability of P2P computing. For example, the Gnutella servent model could not cope with the large connectivity spectrum of the user community, ranging from very low to very high bandwidth. As the Gnutella network grew, it became fragmented because nodes with low bandwidth connections could not keep up with traffic. The idea of *requiring* all functionality to exist at the very edge of the network had to be reconsidered. Eventually, the situation was patched by rendering dumb the low bandwidth servents on the (slow) edges of the network. The notion of centralized reflectors (Gnutella) and super-peers (Morpheus) was (re) invented. A reflector is a powerful high bandwidth gateway for many remote originators with low bandwidth dialup connections. It volunteers to take over the functionality and shield traffic that would normally be carried via low bandwidth servents. However, servents still keep data locally. The agent P2P model naturally covers centralized and decentralized hybrids. Here a powerful node may act as agent for many remote originators. In the remainder of this paper, we follow the agent P2P model and do not use the term *servent* anymore. The terms *originator*, *node* and *agent (node)* are used instead.

2.7 Loop Detection

Query shipping is used to *route* queries through the nodes of the topology. A query remains identical during forwards over hops (unless rewritten or split by a query optimizer). The very same query may arrive at a node multiple times, along distinct routes, perhaps in a complex pattern. Loops in query routes must be detected and prevented. Otherwise, unnecessary or endless multiplication of workloads would be caused. Figure 3 depicts topologies with the potential for a query to become trapped in infinite loops.

To enable loop detection, an originator attaches a different transaction identifier to each query, which is a universally unique identifier (UUID). The transaction identifier always remains identical during query forwarding over hops. A node maintains a state table of recent transaction identifiers and returns an error whenever a query is received that has already been seen. For example, this approach is used in Gnutella.

In practice, it is sufficient for the UUID to be unique with exceedingly large probability, suggesting the use of a 128 bit integer computed by a cryptographic hash digest function such as MD5 [20] or SHA-1 [21] over message text, originator IP address, current time and a random number.

3 Routed vs. Direct Response, Metadata Responses

We propose to distinguish four techniques to return matching query results to an originator: *Routed Response*, *Direct Response*, *Routed Metadata Response*, and *Direct Metadata Response*, as depicted in Figure 4. Let us examine the main implications with a Gnutella use case. A typical Gnutella query such as *"Like a virgin"* is matched by some hundreds of files, most of them referring to replicas of the very same music file. Not all matching files are identical because there exist multiple related songs (e.g. remixes, live recordings) and multiple versions of a song (e.g. with different sampling rates). A music file has a size of at

least several megabytes. Many thousands of concurrent users submit queries to the Gnutella network. A large fraction of users lives on slow and unreliable dialup connections.

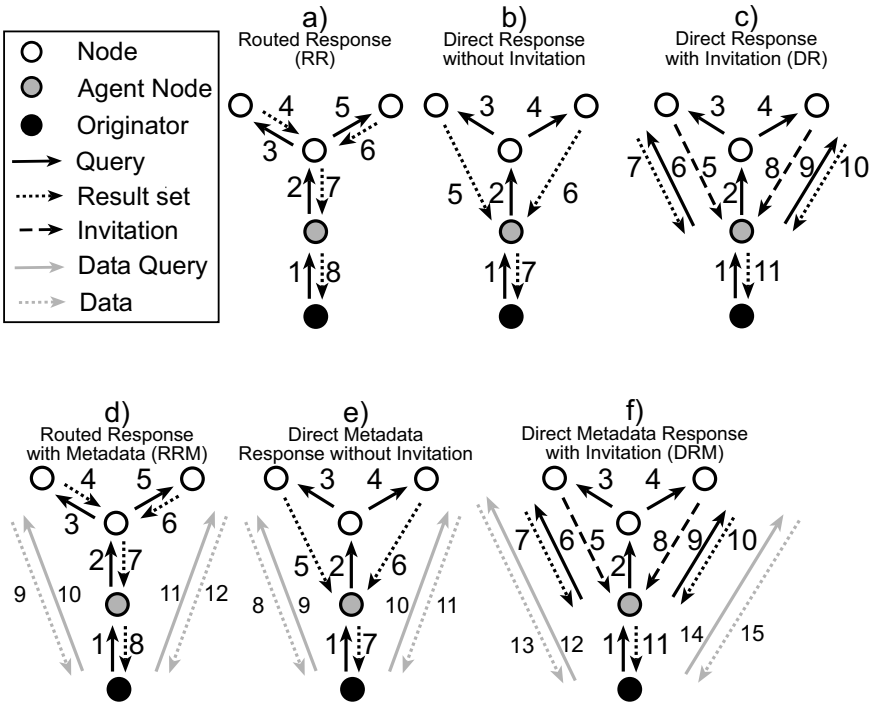

Figure 4: Peer-to-Peer Response Modes.

- **Routed Response.** (Figure 4-a). Results are propagated back into the originator along the paths on which the query flowed outwards. Each (passive) node returns to its (active) client not only its own local results but also all remote results it receives from neighbors. The response protocol is tightly coupled to the query protocol. Routing messages through a logical overlay network of P2P nodes is much less efficient than routing through a physical network of IP routers [22]. Routing back even a single Gnutella file (let alone all results) for each query through multiple nodes would consume large amounts of overall system bandwidth, most likely grinding Gnutella to a screeching halt. As the P2P network grows, it is fragmented because nodes with low bandwidth connections cannot keep up with traffic [23]. Consequently, routed responses are not well suited for file sharing systems such as Gnutella. In general, *overall economics* dictate that routed responses are not well suited for systems that return many and/or large results.

- **Direct Response With and Without Invitation.** To better understand the underlying idea, we first introduce the simpler variant, which is Direct Response Without Invitation (Figure 4-b). Results are not returned by routing back through intermediary nodes. Each (active) node that has local results sends them directly to the (passive) agent, which combines and hands them back to the originator. Response traffic does not travel through the P2P system. It is offloaded via individual point-to-point data transfers on the edges of the network. The response push protocol can be separated from the query protocol. For example, HTTP, FTP or other protocols may be used for response push. Let us examine the main implications with a use case.

As already mentioned, a typical Gnutella query such as *"Like a virgin"* is matched by some hundreds of files, most of them referring to replicas of the very same music file. For Gnutella users it would be sufficient to receive just a small subset of matching files. Sending back *all* such files would unnecessarily consume large amounts of direct bandwidth, most likely restricting Gnutella to users with excessive cheap bandwidth at their disposal. Note however, that the overall Gnutella system would be only marginally affected by a single user downloading, say, a million music files, because the largest fraction of traffic does not travel through the P2P system itself.

In general, *individual economics* dictate that direct responses without invitation are not well suited for systems that return many equal and/or large results, while a small subset would be sufficient. A variant based on invitation (Figure 4-c) softens the problem by inverting control flow. Nodes with matching files do not blindly push files to the agent. Instead they invite the agent to initiate downloads. The agent can then act as it sees fit. For example, it can filter and select a subset of data sources and files and reject the rest of the invitations. Due to its inferiority, the variant without invitation is not considered any further. In the remainder of this thesis, we use the term Direct Response as a synonym for Direct Response With Invitation.

- **Routed Metadata Response and Direct Metadata Response.** Here interaction consists of two phases. In the first phase, routed responses (Figure 4-d) or direct responses (Figure 4-e,f)) are used. However, nodes do not return data results in response to queries, but only small metadata results. The metadata contains just enough information to enable the originator to retrieve the data results and possibly to apply filters before retrieval. In the second phase, the originator selects, based on the metadata, which data results are relevant. The (active) originator directly connects to the relevant (passive) data sources and asks for data results. Again, the largest fraction of response traffic does not travel through the P2P system. It is offloaded via individual point-to-point data transfers on the edges of the network. The retrieval protocol can be separated from the query protocol. For example, HTTP, FTP or other protocols may be used for retrieval.

  The routed metadata response approach is used by file sharing systems such as Gnutella. A Gnutella query does not return files; it just returns an annotated set of HTTP URLs. The originator connects to a subset of these URLs to download files as it sees fit. Another example is a service discovery system where the first phase returns a set of service links instead of full service descriptions. In the second phase, the originator connects to a subset of these service links to download service descriptions as it sees fit. Another example is a *referral* system where the first phase uses routed metadata response to return the service links of the set of nodes having local matching results ("*Go ask these nodes for the answer*"). In the second phase, the originator or agent connects directly to a subset of these nodes to query and retrieve result sets as it sees fit. This variant avoids the "invitation storm" possible under Direct Response. Referrals are also known as *redirections*. A metadata response mode with a radius scope of zero can be used to implement the referral behavior of the Domain Name System (DNS). For details, see Section 7.

3.1 Comparison of Response Mode Properties

Let us compare the properties of the various response models. The following abbreviations are used. RR ... Routed Response, RRM ... Routed Response with metadata, RRX ... Routed Response with and without metadata, DR ... Direct Response, DRX ... Direct Response with and without metadata.

- **Distribution and Location Transparency.** In the response models without metadata, the originator is unaware that (and how) tuples are partitioned among nodes. In other words, these models are transparent with respect to distribution and location. Metadata responses require an originator to contact individual data providers to download full results, and hence are not transparent.

- **(Efficient) Query Support.** All models can answer any query. Both simple and medium queries can be answered efficiently by RRX and DRX, whereas a complex query cannot be answered efficiently. (Justification of this result is deferred to Section 4). Transmission of duplicate results unnecessarily wastes bandwidth. RRX can eliminate duplicates already along the query path, whereas DRX can only do so in the final stage, at the agent. Similarly, maximum result set size limiting is more efficient under RRX because superfluous results can already be discarded along the query path.

- **Economics.** RR results travel multiple hops rather than just a single hop. This leads to poor *overall economics*. The effect is more pronounced for large results, as is the case for music files. RR can also lead to unfortunate *individual economics*. A user that induces few or undemanding queries consumes few system resources. However, if many heavy results for queries from other parties are routed back via such a user's node, it can end up in a situation where it pays for large amounts of bandwidth and gives it away for free to anonymous third parties. For a given user, the costs may drastically outweigh the gains. One could perhaps devise appropriate authorization, quality of service and flow control policies. The unsatisfying economic situation is similar to the one of physical IP routers on the Internet, which also forward traffic from and to third parties. In any case, there remains the fact that results travel multiple hops rather than just one.

  In principle, RRM has the same poor economic properties as RR. However, if metadata is very small in size (e.g. as in Gnutella), then the incurred processing and transmission cost may be acceptable. For example, Gnutella nodes just route back an annotated set of HTTP URLs as metadata. Under DRX, result traffic does not travel through the P2P system. Retrieving results is a deal between just two parties, the provider and the consumer. Consequently, individual economics are controllable and predictable. A user is not charged much for other peoples workloads, unless he explicitly volunteers.

- **Number of TCP Connections at Originator.** Under RR and DR, just one (or no) TCP connection is required at the originator, whereas metadata modes require a connection per (selected) data provider. The more data sources are selected, the more heavyweight data retrieval becomes. Metadata modes can encounter serious latency limitations due to the very expensive nature of secure (and even insecure) TCP connection setup. Hence, the approach does not scale well. However, for many use cases this may not be a problem because a client always selects only a small number of data providers (e.g. 10).

- **Number of TCP Connections at Agent.** Usually a node has few neighbors (five to hundreds). Under RRX, one TCP connection per neighbor is required at an agent. Under DRX, additionally a connection per data provider is required. Again, the more data providers exist, the more heavyweight data retrieval becomes. DRX can encounter serious latency limitations due to the very expensive nature of secure (and even insecure) TCP connection setup. For example, a query that finds the total number of services in the domain `cern.ch` should use RRX. Under DRX, it may generate responses from every single node in that domain. Consequently, an agent can face an invitation storm resembling a denial of service attack. On the other hand, the potential to exploit parallelism is large. All data providers can be handled independently in parallel.

- **Latency.** If a query is of a type that cannot support pipelining (see Section 4.4), the latency for the first result to arrive at the originator is always poor. For a pipelined query, the latency for the first result to arrive is small under DRX, because a response travels a single hop only. Under RRX, a response travels multiple hops, and latency increases accordingly. However, the cost of TCP connection setup at originator and/or agent can invert the situation. Under RR, the cost of TCP connection setup to nodes is paid only once (at node publication time), because connections can typically be kept alive until node deregistration. This is not the case under the other response modes.

- **Caching.** Caching is a technique that trades content freshness for response time. RRX can potentially support caching of content from other nodes at intermediate nodes because response flow naturally concentrates and integrates results from many nodes. DRX nodes return results directly and independently, and hence cannot efficiently support caching.

- **Trust Delegation to Unknown Parties.** Query and result traffic are subject to security attacks. It is not sufficient to establish a secure mutually authenticated channel between any two nodes because malicious nodes can divert routes or modify queries and results. Since a query is almost always routed through multiple hops, many of which are unknown to the agent, we believe that indirect delegation of trust to unknown parties cannot practically be avoided. Security sensitive applications should choose DRX because at least the retrieval of results occurs in a predictable manner between just two parties that can engage in secure mutual authentication and authorization. RRM merely delegates trust on metadata results, but not on full results.

3.2 Response Mode Switches and Shifts

Although from the functional perspective all response modes are equivalent, clearly no mode is optimal under all circumstances. The question arises as to what extent a given P2P network must mandate the use of any particular response mode throughout the system. Observe that nodes are autonomous and defined by their interface only. A node does not "see" what kind of response mode (or technology in general) its neighbors use in answering a query. As long as query semantics are preserved, the node does not care. Consequently, we propose that response modes can be mixed by *switches* and *shifts*, in arbitrary permutations, as depicted in Figure 5.

- **Routed Response ⇒ Direct Response switch.** (Figure 5-a). Starting from the agent, Routed Response is used initially. The central node ("football") receives a query

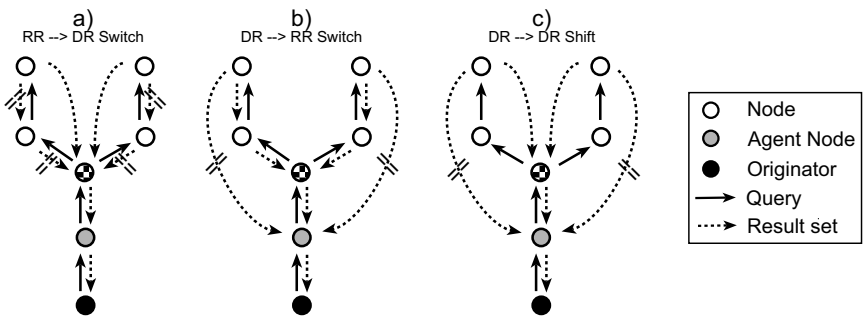

Figure 5: Response Mode Switches and Shifts.

from the agent. For some reason, it decides to answer the query using Direct Response. The response flow that would have been taken under Routed Response is shown crossed out.

- **Direct Response ⇒ Routed Response switch.** (Figure 5-b). Initially, Direct Response is used. However, the "football" decides to answer the query using Routed Response.

- **Direct Response ⇒ Direct Response shift.** (Figure 5-c). Initially, Direct Response is used. The football decides to continue using Direct Response but shift the target of responses. To its own neighbors the football declares itself as (a fake) agent. The responses that would have flowed into the real agent now flow back into the football, and then from the football to the real agent. Note again that this does not break semantics because the football behaves as if the results would have been obtained from its own local database. The real agent receives the same results, but solely from the football.

- **Routed Response ⇒ Routed Response shift.** At each hop, the response target is shifted to be the current node. Interestingly, this kind of shift is at the very heart of the definition of routed response. The classification introduced here shows that this is not the only possible approach.

A node may choose its response mode based on a local and autonomous assessment of the advantages and disadvantages involved. However, because of its context knowledge, often the client (e.g. originator) is in the best position to judge what kind of response mode would be most suitable. Therefore, it is useful to allow specifying as part of the query a hint that indicates the preferred response mode (`routed` or `direct`).

4 Query Processing

In a distributed database system, there exists a single local database and zero or more neighbors. A classic centralized database system is a special case where there exists a single local database and zero neighbors. From the perspective of query processing, a P2P database system has the same properties as a distributed database system, in a recursive structure.

Hence, we propose to organize the P2P query engine like a general distributed query engine [24, 25]. A given query involves a number of operators (e.g. SELECT, UNION, CONCAT,

SORT, JOIN, GROUP, SEND, RECEIVE, SUM, MAX, MAXSETSIZE, IDENTITY) that may or may not be exposed at the query language level. For example, the SELECT operator takes a set and returns a new set with tuples satisfying a given predicate. The UNION operator computes the union of two or more sets. The CONCAT operator concatenates the elements of two or more sets into a list of arbitrary order (without eliminating duplicates). A list can be emulated by a set using distinct surrogate keys. The MAXSETSIZE operator limits the maximum result set size. The IDENTITY operator returns its input set unchanged.

The semantics of an operator can be satisfied by several operator implementations, using a variety of algorithms, each with distinct resource consumption, latency and performance characteristics. The query optimizer chooses an efficient query execution plan, which is a tree plugged together from operators. In an execution plan, a parent operator consumes results from child operators. Query execution is driven from the (final) root consumer in a top down fashion. For example, a request for results from the root operator may in turn lead to a request for results from child operators, which in turn request results from their own child operators, and so on. By means of an execution plan, an optimizer can move a query to data or data to a query. In other words, queries and sub queries can be executed locally or at remote nodes. Performance tradeoffs of query shipping, data shipping and hybrid shipping are discussed in [26].

4.1 Template Query Execution Plan

Recall that Section 2.2 proposed a query model. *Any* query `Q` within our query model can be answered by an agent with the *template execution plan* `A` depicted in Figure 6. The plan applies a local query `L` against the tuple set of the local database. Each neighbor (if any) is asked to return a result set for (the same) neighbor query `N`. Local and neighbor result sets are unionized into a single result set by a unionizer operator `U` that must take the form of either `UNION` or `CONCAT`. A merge query `M` is applied that takes as input the result set and returns a new result set. The final result set is sent to the client, i.e. another node or an originator.

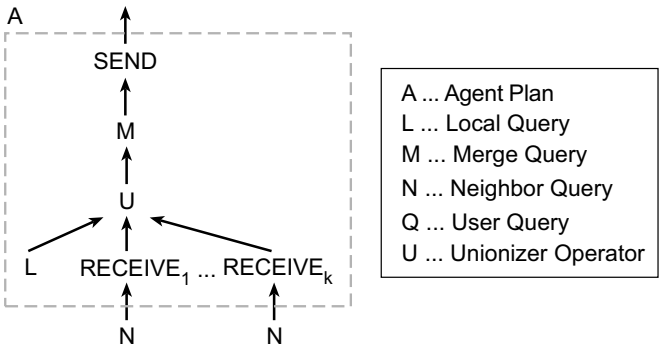

Figure 6: Template Execution Plan.

4.2 Centralized Execution Plan

To see that indeed any query against any kind of database system can be answered within this framework we derive a simple *centralized execution plan* that always satisfies the semantics of any query `Q`. The plan substitutes specific subplans into the template plan `A`, leading to

distinct plans for the agent node (Figure 7-a) and neighbors nodes (Figure 7-b). In the case of XQuery and SQL, parameters are substituted as follows:

XQuery	**SQL**

`A: M = Q`
`U = UNION`
`L = "RETURN /"`
`N' = N`
 	
`N: M = IDENTITY`
`U = UNION`
`L = "RETURN /"`
`N' = N`
	`A: M = Q`
	`U = UNION`
	`L = "SELECT *"`
	`N' = N`
	
	`N: M = IDENTITY`
	`U = UNION`
	`L = "SELECT *"`
	`N' = N`

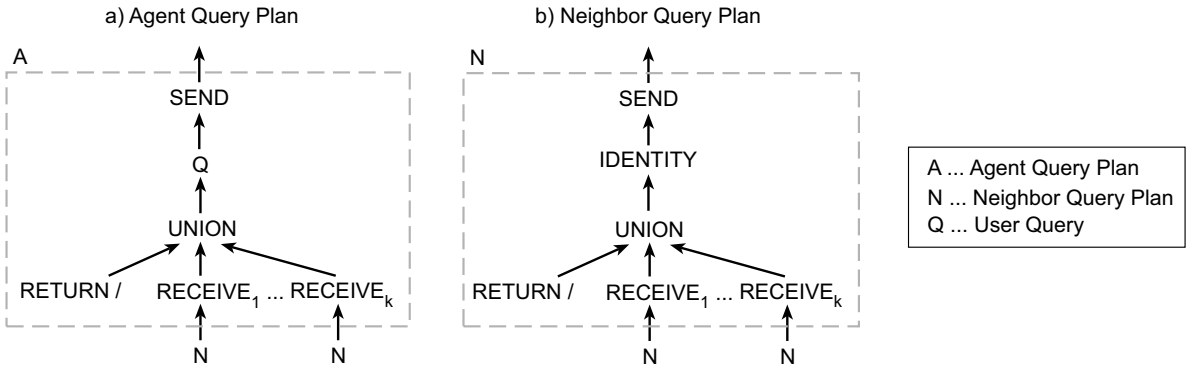

Figure 7: Centralized Execution Plan. The Agent Query Plan (a) fetches all raw tuples from the local and all remote databases, unionizes the result sets, and then applies the query `Q`. Neighbors are handed a rewritten neighbor query (b) that recursively fetches all raw tuples, and returns their union.

In other words, the agent's plan `A` fetches all raw tuples from the local and all remote databases, unionizes the result sets, and then applies the query `Q`. Neighbors are handed a rewritten neighbor query `N` that recursively fetches all raw tuples, and returns their union. The neighbor query `N` is recursively partitionable (see below).

The same centralized plan works for routed and direct response, both with and without metadata. Under direct response, a node does forward the query `N`, but does not attempt to receive remote result sets (conceptually empty result sets are delivered). The node does not send a result set to its predecessor, but directly back to the agent.

In a distributed database system, there exists a single local database and zero or more neighbors. A classic centralized database system is a special case where there exists a single local database and zero neighbors. From the perspective of query processing, a P2P database system has the same properties as a distributed database system, in a recursive structure. Consequently, the very same centralized execution plan applies to any kind of database system; and any query within our query model can be answered.

The centralized execution plan can be inefficient because potentially large amounts of base data have to be shipped to the agent before locally applying the user's query. However, sometimes this is the only plan that satisfies the semantics of a query. This is always the case

for a complex query. A more efficient execution plan can sometimes be derived (as proposed below). This is always the case for a simple and medium query.

4.3 Recursively Partitionable Query

A P2P network can be efficient in answering queries that are recursively partitionable. A query `Q` is *recursively partitionable* if, for the template plan `A`, there exists a merge query `M` and a unionizer operator `U` to satisfy the semantics of the query `Q` assuming that `L` and `N` are chosen as `L = Q` and `N = A`. In other words, a query is recursively partitionable if the very same execution plan *can* be recursively applied at every node in the P2P topology. The corresponding execution plan is depicted in Figure 8.

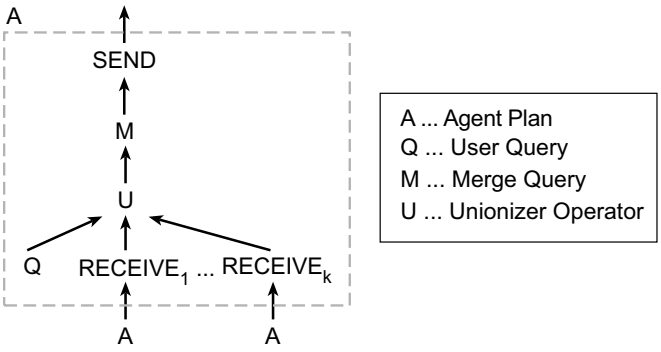

Figure 8: Execution Plan for Recursively Partitionable Query.

The input and output of a merge query have the same form as the output of the local query `L`. Query processing can be parallelized and spread over all participating nodes. Potentially very large amounts of information can be searched while investing little resources such as processing time per individual node. The recursive parallel spread of load implied by a recursively partitionable query is the basis of the massive P2P scalability potential. However, query performance is not necessarily good, for example due to high network I/O costs.

Now we are in the position to clarify the definition of simple, medium and complex queries.

- *Simple Query.* A query is *simple* if it is recursively partitionable using `M = IDENTITY, U = UNION`.
- *Medium Query.* A query is a *medium* query if it is not simple, but it is recursively partitionable.
- *Complex Query.* A query is *complex* if it is not recursively partitionable.

It is an interesting open question (at least to us) if a query processor can automatically determine whether a correct merge query and unionizer exist, and if so, how to choose them. Related problems have been studied extensively in the context of distributed and parallel query processing as well as query rewriting for heterogeneous and homogenous relational database systems [24, 27, 25]. Distributed XQueries are an emerging field [28]. For simplicity, in the remainder of this paper we assume that the user explicitly provides `M` and `U` along with a query `Q`. If `M` and `U` are not provided as part of a query to any given node, the node acts defensively by assuming that the query is not recursively partitionable. Choosing `M` and `U` is straightforward for a human being. Consider for example the following medium XQueries.

- *Return the number of replica catalog services.* The merge query computes the sum of a set of numbers. The unionizer is CONCAT.

```
Q = RETURN
      <tuple>
        count(/tupleset/tuple/content/service[interface/@type="repcat"])
      </tuple>
M = RETURN
      <tuple>
        sum(/tupleset/tuple)
      </tuple>
U = CONCAT
```

- *Find the service with the largest uptime.*

```
Q=M= RETURN (/tupleset/tuple[@type="service"] SORTBY (./@uptime)) [last()]
U =  UNION
```

Note that the query engine always encapsulates the query output with a `tupleset` root element. A query need not generate this root element as it is implicitly added by the environment.

A custom merge query can be useful. For example, assume that each individual result tuple is tagged with a timestamp indicating the time when the information expires and ceases to be valid. A custom merge query can ignore all results that have already expired. Alternatively, it can ignore all results but the one with the most recent timestamp. As another example, a custom merge query can cut off all but the first 100 result tuples. Such a result set size limiting feature (`maxResults`) attempts to make bandwidth consumption more predictable.

4.4 Pipelining

The success of many applications depends on how fast they can start producing initial/relevant portions of the result set rather than how fast the entire result set is produced [29]. This is particularly often the case in distributed systems where many nodes are involved in query processing, each of which may be unresponsive for many reasons. The situation is even more pronounced in systems with loosely coupled autonomous nodes.

Often an originator would be happy to already do useful work with one or a few *early results*, as long as they arrive quickly and reliably. Results that arrive later can be handled later, or are ignored anyway. For example, in an interactive session, a typical Gnutella user is primarily interested in being able to start *some* music download as soon as possible. The user is quickly disappointed when not a single result for a query arrives in less than four seconds. Choosing among 1000 species of "Like a virgin" is interesting, but helps little if it comes at the expense of, say, one minute idle waits. As another example, consider a user that wants to discover schedulers to submit a job. It is interesting to discover that 100 schedulers are available, but the primary requirement is to find at least three quickly and reliably.

Database theory and practice establishes that query execution engines in general, and distributed query execution engines in particular should be based on iterators [24]. An operator corresponds to an iterator class. Iterators of any kind have a uniform interface, namely the three methods `open()`, `next()` and `close()`. In an execution plan, a parent iterator consumes results from child iterators. Query execution is driven from the (final) root consumer in a top down fashion. For example, a call to `next()` may call `next()` on child

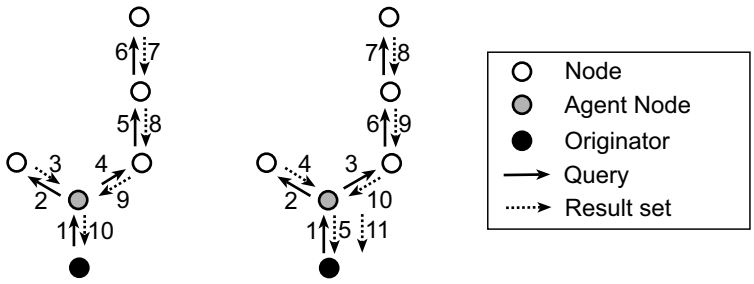

Figure 9: Non-Pipelined (left) and Pipelined Query (right).

Query Type	Supports Pipelining?
Simple Query	Yes
Medium Query	Maybe
Complex Query	Typically No

Table 1: Pipelining Support of Query Types.

iterators, which in turn call `next()` on their child iterators, and so on. For efficiency, the method `next()` can be asked to deliver several results at once in a so-called *batch*. Semantics are as follows: *"Give me a batch of at least* `N` *and at most* `M` *results"* (less than `N` results are delivered when the entire query result set is exhausted). For example, the SEND and RECEIVE network communication operators (iterators) typically work in batches.

The monotonic semantics of certain operators such as SELECT, UNION, CONCAT, SEND, RECEIVE, MAXSETSIZE, IDENTITY allow that operator implementations consume just one or a few child results on `next()`. In contrast, the non-monotonic semantics of operators such as SORT, GROUP, MAX, some JOIN methods, etc. require that operator implementations consume *all* child results already on `open()` in order to be able to deliver a result on the first call to `next()`. Since the output of these operators on a subset of the input is not, in general, a subset of the output on the whole input, these operators need to see all of their input before they produce the correct output. This does not break the iterator concept but has important latency and performance implications. Whether the root operator of an agent exhibits a short or long latency to deliver to the originator the first result from the result set depends on the query operators in use, which in turn depend on the given query. In other words, for some query types the originator has the potential to immediately start piping in results (at moderate performance rate), while for other query types it must wait for a long time until the first result becomes available (the full result set arrives almost at once, however).

A query (an operator implementation) is said to be *pipelined* if it can already produce at least one result tuple before all input tuples have been seen. Otherwise, a query (an operator) is said to be *non-pipelined*. Figure 9 depicts examples for both modes.

Simple queries do support pipelining (e.g. Gnutella queries). Medium queries may or may not support pipelining, whereas complex queries typically do not support pipelining. The properties are summarized in Table 1.

Bear in mind, that even if a query can be pipelined, the messaging model and underlying network layers in use may not support pipelining, in which case a result set has to be delivered with long latency in a single large batch. Finally note that non-pipelining delivery without a dynamic abort timeout feature is highly unreliable due to the so-called simultaneous abort

problem (see below). If only one of the many nodes in the query path fails to be responsive for whatever reasons, all other nodes in the chain are waiting, eventually time out at the same time, and the originator receives not even a single result.

5 Static Loop Timeout and Dynamic Abort Timeout

Clearly there comes a time when a user is no longer interested in query results, no matter whether any more results might be available. The query roaming the network and its response traffic should fade away after some time. In addition, P2P systems are well advised to attempt to limit resource consumption by defending against *runaway* queries roaming forever or producing gigantic result sets, either unintended or malicious. To address these problems, an absolute *abort timeout* is attached to a query, as it travels across hops. An abort timeout can be seen as a deadline. Together with the query, a node tells a neighbor *"I will ignore (the rest of) your result set if I have not received it before 12:00:00 today."* The problem, then, is to ensure that a maximum of results can be delivered reliably within the time frame desired by a user.

The value of a *static timeout* remains unchanged across hops, except for defensive modification in flight triggered by runaway query detection (e.g. infinite timeout). In contrast, it is intended that the value of a *dynamic timeout* be decreased at each hop. Nodes further away from the originator may time out earlier than nodes closer to the originator.

5.1 Dynamic Abort Timeout

A static abort timeout is entirely unsuitable for non-pipelined result set delivery, because it leads to a serious reliability problem, which we propose to call *simultaneous abort timeout.* If just one of the many nodes in the query path fails to be responsive for whatever reasons, all other nodes in the path are waiting, eventually time out and attempt to return at least a partial result set. However, it is impossible that any of these partial results ever reach the originator, because all nodes time out *simultaneously* (and it takes some time for results to flow back). For example, the agent times out and attempts to return its local partial results to the originator. After that, all partial results flowing to the agent from neighbors, and their neighbors, etc. are discarded – it is already too late. However, even the agent cannot deliver results to the originator because the originator has already timed out (shortly) before the results arrive. Hence, the originator receives not even a single result if just one of the many nodes in the query path fails to be responsive.

To address the simultaneous abort timeout problem, we propose dynamic abort timeouts. Under *dynamic abort timeout*, nodes do not time out at the same time. Instead, nodes further

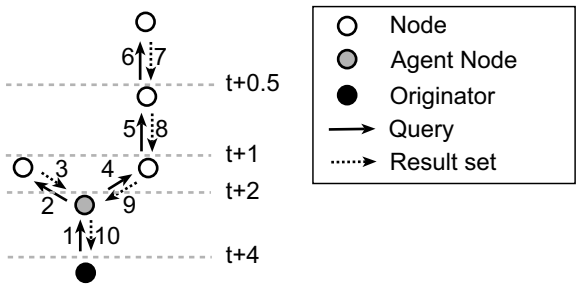

Figure 10: Dynamic Abort Timeout.

away from the originator time out earlier than nodes closer to the originator. This provides some safety time window for the partial results of any node to flow back across multiple hops to the originator. Together with the query, a node tells a neighbor *"I will ignore (the rest of) your result set if I have not received it before 12:00:00 today. Do whatever you think is appropriate to meet this deadline"*. Intermediate nodes can and should adaptively decrease the timeout value as necessary, in order to leave a large enough time window for receiving and returning partial results subsequent to timeout.

Observe that the closer a node is to the originator, the more important it is (because if it cannot meet its deadline, results from a large branch are discarded). Further, the closer a node is to the originator, the larger is its response and bandwidth consumption. Thus, as a good policy to choose the safety time window, we propose *exponential decay with halving*. The window size is halved at each hop, leaving large safety windows for important nodes and tiny window sizes for nodes that contribute only marginal result sets. Also, taking into account network latency and the time it takes for a query to be locally processed, the timeout is updated at each hop `N` according to the following recurrence formula:

$$timeout_N = currenttime_N + \frac{timeout_{N-1} - currenttime_N}{2} \quad (1)$$

Consider for example Figure 10. At time `t` the originator submits a query with a dynamic abort timeout of `t+4` seconds. In other words, it warns the agent to ignore results after time `t+4`. The agent in turn intends to safely meet the deadline and so figures that it needs to retain a safety window of 2 seconds, already starting to return its (partial) results at time `t+2`. The agent warns its own neighbors to ignore results after time `t+2`. The neighbors also intend to safely meet the deadline. From the 2 seconds available, they choose to allocate 1 second, and leave the rest to the branch remaining above. Eventually, the safety window becomes so small that a node can no longer meet a deadline on timeout. The results from the unlucky node are ignored, and its partial results are discarded. However, other nodes below and in other branches are unaffected. Their results survive and have enough time to hop all the way back to the originator before time `t+4`.

Instead of ignoring results which miss their deadline a node may also close the connection. This may, but need not, be harmless. The connection is typically simply reestablished as soon as a new query is to be forwarded. However, in an attempt to educate good P2P citizens, a node may choose to stop propagating or deny service to neighbors that repeatedly do not meet abort deadlines. For example, a strategy may use an exponential back-off algorithm. Note that as long as a node obeys its timeout it can independently implement any timeout policy it sees fit for its purposes without regard to the policy implemented at other nodes. If a node misbehaves or maliciously increases the abort timeout, it risks not being able to meet its own deadline, and is likely soon dropped or denied service. Such healthy measures move less useful nodes to the edge of the network where they cause less harm, because their number of topology links tends to decrease.

To summarize, under non-pipelined result set delivery, dynamic abort timeouts using *exponential decay with halving* ensure that a maximum of results can be delivered reliably within the time frame desired by a user. We speculate that dynamic timeouts could also incorporate sophisticated cost functions involving latency and bandwidth estimation and/or economic models.

5.2 Static Loop Timeout

Interestingly, a static loop timeout is required in order to fully preserve query semantics. A dynamic timeout (e.g. the dynamic abort timeout) is unsuitable to be used as loop timeout. Otherwise, a problem arises that we propose to call *non-simultaneous loop timeout*. Recall from Section 2.7 that the same query may arrive at a node multiple times, along distinct routes, perhaps in a complex pattern. Loops in query routes must be detected and prevented. Otherwise, unnecessary or endless multiplication of workloads would be caused. To this end, a node maintains a state table of recent transaction identifiers and associated *loop timeouts* and returns an error whenever a query is received that has already been seen (according to the state table). Before the loop timeout is reached, the same query can potentially arrive multiple times, along distinct routes. On loop timeout, a node may "forget" about a query by deleting it from the state table. To be able to reliably detect a loop, a node must not forget a transaction identifier before its loop timeout has been reached.

However, let us assume for the moment that a dynamic timeout (e.g. the dynamic abort timeout) is used as loop timeout. Consider for example, Figure 11, which is identical to Figure 10 except that the agent has an additional neighbor that can potentially receive the query along more than one path. At time `t` the originator submits a query with a dynamic abort timeout of `t+4` seconds. The agent in turn warns its own neighbors to ignore results after time `t+2`. Request 5 is sent and arrives, is processed, and its results (step 8) are delivered before the dynamic abort timeout of time `t+1`. At time `t+1` the loop timeout is reached and the query is deleted from the state table. For many reasons, including temporary network segment problems and sequential neighbor processing, request 10 can be delayed. In the example it arrives after time `t+1`. By this time, the receiving node has already forgotten that it already handled the very same query. Hence, the node cannot detect the loop and continues to process and forward (step 11, 12) the same query again.

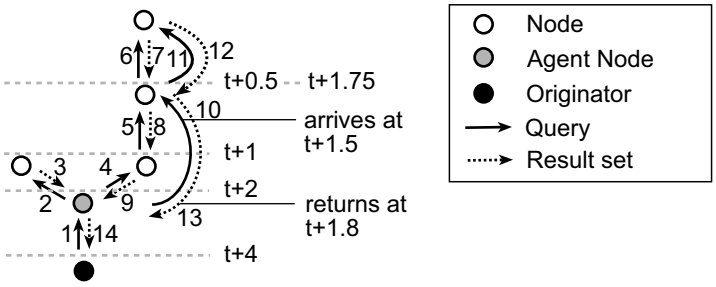

Figure 11: Loop Detection Failure with Dynamic Loop Timeout.

The non-simultaneous loop timeout problem is caused by the fact that some nodes still forward the query to other nodes when the destinations have already forgotten it. In other words, the problem is that loop timeout does not occur simultaneously everywhere. Consequently, a loop timeout must be static (does not change across hops) to guarantee that loops can reliably be detected. Along with a query, an originator not only provides a dynamic abort timeout, but also a static loop timeout. Initially at the originator, both values must be identical (e.g. `t+4`). After the first hop, both values become unrelated.

To summarize, we have `abort timeout` $\leq$ `loop timeout`. Loop timeouts must be static whereas abort timeouts may be static or dynamic. Under non-pipelined result set delivery, dynamic abort timeouts using *exponential decay with halving* ensure that a maximum of results can be delivered reliably within the time frame desired by a user. A dynamic

abort timeout model still requires static loop timeouts to ensure reliable loop detection, so that a node does not forward and answer the same query multiple times.

6 Query Scope

As in a data integration system, the goal is to exploit several independent information sources as if they were a single source. This is important for distributed systems in which node topology or deployment model change frequently. For example, cross-organizational Grids and P2P networks exhibit such a character. However, in practice, it is often sufficient (and much more efficient) for a query to consider only a subset of all tuples (service descriptions) from a subset of nodes. For example, a typical query may only want to search tuples (services) within the scope of the domain `cern.ch` and ignore the rest of the world.

Recall that to this end, Section 2.2 cleanly separated the concepts of (logical) query and (physical) query scope. A query is formulated against a global database view and is insensitive to link topology and deployment model. In other words, to a query the set of tuples appears as a single homogenous database, even though the set may be (recursively) partitioned across many nodes and databases. This means that in a relational or XML environment, at the global level, the set of all tuples appears as a single, very large, table or XML document, respectively. The query scope, on the other hand, is used to navigate and prune the link topology and filter on attributes of the deployment model. Conceptually, the scope is the input fed to the query. The query scope is a set and may contain anything from all tuples in the universe to none. Both query and scope can prune the search space, but they do so in a very different manner.

A query scope is specified either *directly* or *indirectly*. For example, one can directly enumerate the tuples (service descriptions) to be considered. However, this is usually impractical. One can also indirectly define a query scope by specifying a set of nodes, implying that the query should be evaluated against the union of all tuples contained in their respective databases. One can distinguish scopes based on neighbor selection, timeout and radius. Note that for security reasons a node may choose to ignore or override a third party provided query scope, for example to guard against runaway queries with infinite scope.

6.1 Neighbor Selection

For simplicity, all our discussions so far have implicitly assumed a *broadcast* model (on top of TCP) in which a node forwards a query to all neighbor nodes. However, in general one can select a subset of neighbors, and forward concurrently or sequentially. Fewer query forwards lead to less overall resource consumption. The issue is critical because of the snowballing (epidemic, flooding) effect implied by broadcasting. Overall bandwidth consumption grows exponentially with the query radius, producing enormous stress on the network and drastically limiting its scalability. For details, consult [30, 22].

Clearly selecting a neighbor subset can lead to incomplete coverage, missing important results. The best policy to adopt depends on the context of the query and the topology. Context is *required* to improve on the broadcast model. For example, it makes little sense to forward a Gnutella query to non-Gnutella nodes. The scope can select only neighbors with a service description of interface type "Gnutella". In an attempt to explicitly exploit topology characteristics, a virtual organization of a Grid may deliberately organize global, intermediate and local job schedulers into a tree-like topology. Correct operation of scheduling may require reliable discovery of all or at least most relevant schedulers in the tree. In such a scenario,

random selection of half of the neighbors at each node is certainly undesirable. A policy that selects all `child` nodes and ignores all `parent` nodes may be more adequate.

Further, a node may maintain statistics about its neighbors. One may only select neighbors that meet minimum requirements in terms of latency, bandwidth or historic query outcomes (`maxLatency, minBandwidth, minHistoricResult`). Other node properties such as hostname, domain name, owner, etc. can be exploited in query scope guidance, for example to implement security policies. Consider an example where the scheduling system may only trust nodes from a select number of security domains. Here a query should never be forwarded to nodes not matching the trust pattern.

Further, in some systems, finding a single result is sufficient. In general, a user or any given node can guard against unnecessarily large result sets, message sizes and resource consumption by specifying the maximum number of result tuples (`maxResults`) and bytes (`maxResultsBytes`) to be returned. Using sequential propagation, depending on the number of results already obtained from the local database and a subset of the selected neighbors, the query may no longer need to be forwarded to the rest of the selected neighbors.

6.2 Neighbor Selection Query

For flexibility and expressiveness, we propose to allow the user to specify the selection policy. In addition to the normal query, the user defines a *neighbor selection query* (XQuery) that takes the tuple set of the current node as input and returns a subset that indicates the nodes selected for forwarding. For example, a neighbor query implementing broadcasting selects all services with registry and P2P query capabilities, as follows:

```
RETURN /tupleset/tuple[@type="service"
  AND content/service/interface[@type="Consumer-1.0"]
  AND content/service/interface[@type="XQuery-1.0"]]
```

A wide range of policies can be implemented in this manner. The neighbor selection policy can draw from the rich set of information contained in the tuples published to the node. Tuple metadata such as type, context, timestamps, etc. can be used for neighbor selection decisions (see Section 2.2). Further, recall that the set of tuples in a database may not only contain service descriptions of neighbor nodes (e.g. in WSDL [18] or SWSDL [2]), but also other kind of content published from any kind of content provider. For example, this may include host and network information as well as statistics a node periodically publishes to its immediate neighbors.

For example, broadcast and random selection can be expressed with a neighbor query. One can select nodes that support given interfaces (e.g. Gnutella [19], Freenet [31] or job scheduling). In a tree topology, a policy can use the tuple `context` attribute to select all `child` nodes and to ignore all `parent` nodes. One can implement domain filters and security filters (e.g. `allow/deny` regular expressions as used in the Apache HTTP server [32]) if the tuple set includes metadata such as hostname and node owner. Power-law policies [33] can be expressed if metadata includes the number of neighbors to the `n`-th radius.

As usual, for security reasons, a node may choose to ignore, override or extend any scope hints it receives. The neighbor query concept can also be used for flexible policy implementation internal to a node. In this case, a node always ignores the user provide neighbor query and uses an internal custom neighbor selection query instead.

6.3 Timeout

Clearly there comes a time when a user is no longer interested in query results, no matter whether any more results might be available. The query roaming the network and its response traffic should fade away after some time. Section 5 already discussed in depth this issue and its implications on loop detection and non-pipelined result set delivery. Here we just note that timeouts clearly belong to the query scope, rather than the query itself.

6.4 Radius

The *radius* of a query is a measure of path length. More precisely, it is the maximum number of hops a query is allowed to travel on any given path. The radius is decreased by one at each hop. The roaming query and response traffic must fade away upon reaching a radius of less than zero. A scope based on radius serves similar purposes as a timeout. Nevertheless, timeout and radius are complementary scope features. The radius can be used to indirectly limit result set size. In addition, it helps to limit latency and bandwidth consumption and to guard against runaway queries with infinite lifetime. In Gnutella and Freenet, the radius is the primary means to specify a query scope. The radius is termed *TTL (time-to-live)* in these systems. Neither of these systems support timeouts.

For maximum result set size limiting, a timeout and/or radius can be used in conjunction with neighbor selection, routed response, and perhaps sequential forward, to implement the *expanding ring* [34] strategy. The term stems from IP multicasting. Here an agent first forwards the query to a small radius/timeout. Unless enough results are found, the agent forwards the query again with increasingly large radius/timeout values to reach further into the network, at the expense of increasingly large overall resource consumption. On each expansion radius/timeout are multiplied by some factor.

We now turn to some more subtle points. When precisely does the radius trigger the end of query life? A node rejects a query with a radius less than zero. When a node accepts a query, the radius is decreased by one, and the query is evaluated. The query is not forwarded to neighbors if the new radius is less than zero. In other words, a new radius less than zero forces a neighbor selection policy that yields an empty set. For example, an originator can determine the neighbors of an agent by sending it a query with a radius of zero hops.

Note that the radius is not defined to be *the number* of hops a query is allowed to travel on any given path. Rather, it is more weakly defined to be the maximum number of hops a query is allowed to travel on any given path. In other words, it is not guaranteed that a query takes the shortest path from the agent to any given node, thereby covering a total maximum of nodes. There are two reasons for this kind of definition. First, a node may choose to decrease the radius by any value it sees fit in order to reduce resource consumption or to prevent system exploitation. Second, loop detection and unpredictable timing in distributed systems can lead to a phenomenon we propose to call *greedy radius pruning*. Recall that the very same query may arrive at a node multiple times, along distinct routes, perhaps in a complex pattern. Traveling `N` hops decreases the radius of a query by `N`. If the query first arrives via a route with many (fast) hops, and later arrives again via a route with few (slow hops), the second arrival will be detected as a loop and rejected. However, the successfully forwarded (first) query continues to travel less hops than theoretically possible considering the (larger) radius of the second query. If the second query had arrived first, the query would have been able to travel further and potentially collect more matching results. Propagating a query to all neighbors concurrently may somewhat increase query coverage, in particular in homogenous LANs.

7 Related Work

Agent P2P Model. The underlying idea of the Agent P2P model is not new. Consider for example, the email infrastructure model [35]. Typically, a single central high availability agent serves outgoing and incoming mail for originators from an entire organization. However, an email system can also be fully (or partly) decentralized such that each originator runs its own agent on its own host. This transparent flexibility contributes to the widespread adoption and tremendous success of email as an Internet "killer application". A similar example is the X.500 [36] directory architecture, which has a *Directory User Agent* (originator) querying a *Home Directory System Agent* (agent node), which is one of a collection of *Directory System Agents* (nodes).

Loop Detection. The X.500 protocol [36] uses a route-tracing algorithm for loop detection. This algorithm only works for queries that select on a name from a hierarchical name space that is mimicked by the link topology. In our general context, the algorithm is insufficient for loop detection because we allow, but do not assume, such a topology and namespace. The route-tracing algorithm attaches to a query the route already taken, represented by a list of node identifiers. On query forward, a node N appends its own identifier to the route. A loop is detected if the identifier of the current node N is already contained in the route. This mechanism only detects a loop if a query forwarded by a given node N eventually arrives again at the same node N. It cannot detect the more common form of loop where the same query arrives along multiple paths at a given node N, but none of the paths have so far touched N.

Query Processing. None of the example discovery queries from Section 2.2 can be satisfied with a lookup by key (e.g. globally unique name). This is the type of query assumed in most P2P systems such as DNS [37], Gnutella [19], Freenet [31], Tapestry [38], Chord [39] and Globe [40], leading to highly specialized *content-addressable* networks centered around the theme of distributed hash table lookup. Note further that almost no queries are exact match queries (i.e. given a flat set of attribute values find all tuples that carry exactly the same attribute values), assumed in systems such as SDS [41] and Jini [42]. Our approach is distinguished in that it not only supports all of the above query types, but it also supports queries from the rich and expressive general-purpose query languages XQuery [12] and SQL [13].

Pipelining. For a survey of adaptive query processing, including pipelining, see the special issue of [43]. [44] develops a general framework for producing partial results for queries involving any non-monotonic operator. The approach inserts update and delete directives into the output stream. The Tukwila [45] and Niagara projects [46] introduce data integration systems with adaptive query processing and XML query operator implementations that efficiently support pipelining. Pipelining of hash joins is discussed in [47, 48, 49]. [50] proposes a rate based pipeline scheduling algorithm that prioritizes and schedules the flow of data between pipelined operators so that the result output rate is maximized. The algorithm is also demonstrated with pipelined hash joins. Pipelining is sometimes also termed *streaming* or *non-blocking* execution.

Neighbor Selection. *Iterative deepening* [51] is a similar technique to *expanding ring* where an optimization is suggested that avoids reevaluating the query at nodes that have already done so in previous iterations. Neighbor selection policies that are based on randomness and/or historical information about the result set size of prior queries are simulated and analyzed in [52]. An efficient neighbor selection policy is applicable to simple queries posed to networks in which the number of links of nodes exhibits a power law distribution (e.g.

Freenet and Gnutella) [33]. Here most (but not all) matching results can be reached with few hops by selecting just a very small subset of neighbors (the neighbors that themselves have the most neighbors to the n-th radius). Note, however, that the policy is based on the assumption that not all results must be found and that all query results are equally relevant. Depending on the application context, this assumption may or may not be valid. These related works discuss in isolation neighbor selection techniques for a particular query type, without the context of a framework for comprehensive query support.

DNS. Distributed databases with a hierarchical name space such as the Domain Name System (DNS) [37] can efficiently answer queries of the form *"Find an object by its full name"*. For example, the DNS can search for the IP address (e.g. `137.138.29.51`) of a given domain name (e.g. `fred.cms.cern.ch`). Because of the nature of the supported query type, these systems arrange the link topology, according to the hierarchical name space, as a tree topology, as depicted in Figure 12. Each node (DNS server) is responsible for tuples from a name space sub-tree such as `ch`, `cern.ch` or `cms.cern.ch`. A node may internally partition its name space and delegate responsibility for sub-trees to child nodes. A node holds a database of tuples, each of which carries a domain name and an IP address.

A query searching for the IP address of a domain name traverses the tree on the shortest path from originator (e.g. `fire.ethz.ch`) to the node containing the domain name - first up, then down. At each node, a *name resolution* policy selects the neighbor "closer" to the name than the current node, according to name space metadata. In DNS, queries are not forwarded (routed) through the topology. Instead, a node returns a *referral* message that redirects an originator to the next closer node. The originator explicitly queries the next node, is referred to yet another closer node, and so on. Nodes cache the IP address (service link result) of recently queried nodes. A cache hit directly refers a client to the responsible node. This dramatically reduces load on nodes close to the root of the hierarchy. It also reduces the number of round trips involved for a query.

To support neighbor selection in a hierarchical name space within our UPDF framework, a node could publish to its neighbors not only its service link, but also the name space it manages. A natural candidate for the hierarchical name is the content link of a tuple. The DNS referral behavior can be implemented within UPDF by using a radius scope of zero. The same holds for the LDAP referral behavior (see below).

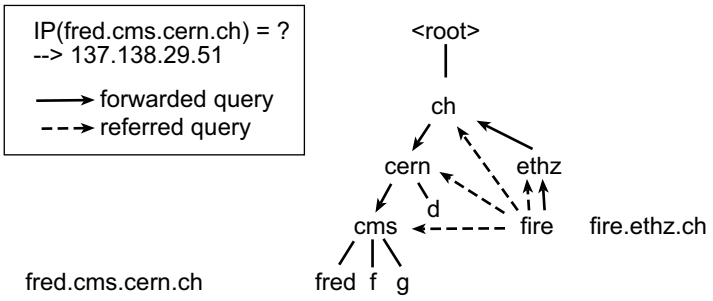

Figure 12: Query Flow in Domain Name System (DNS).

X.500, LDAP and MDS. The hierarchical distributed X.500 directory [36] works similarly to the DNS. It also supports referrals, but in addition can forward queries through the topology (*chaining* in X.500 terminology). The query language is simple [2]. Route tracing is used as a loop detection algorithm. Query scope specification can support maximum result set size limiting. It does not support radius and dynamic abort timeout as well as

pipelined query execution across nodes. LDAP [53] is a simplified subset of X.500. Like DNS, it supports referrals but not query forwarding. The Metacomputing Directory Service (MDS) [54, 55] inherits all properties of LDAP. MDS additionally implements a simple form of query forwarding that allows for multi-level hierarchies but not for arbitrary topologies. Here neighbor selection forwards the query to LDAP servers overlapping with the query name space. The query is forwarded "as is", without loop detection. Further, MDS does not support radius and dynamic abort timeout, pipelined query execution across nodes as well as direct response and metadata responses.

8 Conclusions

We take the first steps towards unifying the fields of database management systems and P2P computing, which so far have received considerable, but separate, attention. We extend database concepts and practice to cover P2P search. Similarly, we extend P2P concepts and practice to support powerful general-purpose query languages such as XQuery and SQL. As a result, we propose the so-called *Unified Peer-to-Peer Database Framework (UPDF)* for general-purpose query support in large heterogeneous distributed systems spanning many administrative domains. UPDF is unified in the sense that it allows to express specific applications for a wide range of data types, node topologies, query languages, query response modes, neighbor selection policies, pipelining characteristics, timeout and other scope options. The uniformity, wide applicability and reusability of our approach distinguish it from related work, which individually addresses some but not all problem areas.

Traditional distributed systems assume a particular type of topology (e.g. hierarchical as in DNS, LDAP). Existing P2P systems are built for a single application and data type and do not support queries from a general-purpose query language. For example, Gnutella, Freenet, Tapestry, Chord, Globe and DNS only support lookup by key (e.g. globally unique name). Others such as SDS, LDAP and MDS support simple special-purpose query languages, leading to special-purpose solutions unsuitable for multi-purpose service and resource discovery in large heterogeneous distributed systems spanning many administrative domains. [33] discusses in isolation neighbor selection techniques for a particular query type, without the context of a framework for comprehensive query support. LDAP and MDS do not support essential features for P2P systems such as reliable loop detection, non-hierarchical topologies, dynamic abort timeout, query pipelining across nodes as well as radius scoping. None introduce a unified P2P database framework for general-purpose query support.

The results presented in this paper open two interesting research directions. First, it would be interesting to extend further the unification and extension of concepts from Database Management Systems and P2P computing. For example, one could consider the application of database techniques such as buffer cache maintenance, view materialization, placement and selection as well as query optimization for use in P2P computing. These techniques would need to be extended in the light of the complexities stemming from autonomous administrative domains, inconsistent and incomplete (soft) state, dynamic and flexible cache freshness policies and, of course, tuple updates. An important problem left open in our work is the question if a query processor can automatically determine whether a correct merge query and unionizer exist, and if so, how to choose them. Here approaches from query rewriting for heterogeneous and homogenous relational database systems [24, 27] should prove useful. Further, database resource management and authorization mechanisms might be worthwhile to consider for specific flow control policies per query or per user.

Second, it would be interesting to study and specify a messaging and communication

model, as well as network protocol that uniformly supports P2P database queries for a wide range of database architectures and response models such that the stringent demands of ubiquitous Internet infrastructures in terms of interoperability, extensibility, reliability, efficiency and scalability can be met. Here, issues of high concurrency, low latency as well as early and/or partial result set retrieval need to be addressed. Further, resource consumption and flow control should be encouraged on a per query basis.

References

[1] Large Hadron Collider Committee. Report of the LHC Computing Review. Technical report, CERN/LHCC/2001-004, April 2001. http://lhc-computing-review-public.web.cern.ch/lhc-computing-review-public/Public/Report_final.PDF.

[2] Wolfgang Hoschek. *A Unified Peer-to-Peer Database Framework for XQueries over Dynamic Distributed Content and its Application for Scalable Service Discovery*. PhD Thesis, Technical University of Vienna (submitted), 2002.

[3] Ben Segal. Grid Computing: The European Data Grid Project. In *IEEE Nuclear Science Symposium and Medical Imaging Conference*, Lyon, France, October 2000.

[4] Wolfgang Hoschek, Javier Jaen-Martinez, Asad Samar, Heinz Stockinger, and Kurt Stockinger. Data Management in an International Data Grid Project. In *1st IEEE/ACM Int. Workshop on Grid Computing (Grid'2000)*, Bangalore, India, December 2000.

[5] Dirk Düllmann, Wolfgang Hoschek, Javier Jean-Martinez, Asad Samar, Ben Segal, Heinz Stockinger, and Kurt Stockinger. Models for Replica Synchronisation and Consistency in a Data Grid. In *10th IEEE Symposium on High Performance and Distributed Computing (HPDC-10)*, San Francisco, California, August 2001.

[6] Ian Foster, Carl Kesselman, and Steve Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Int. Journal of Supercomputer Applications*, 15(3), 2001.

[7] Ian Foster, Carl Kesselman, Jeffrey Nick, and Steve Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, January 2002.

[8] P. Cauldwell, R. Chawla, Vivek Chopra, Gary Damschen, Chris Dix, Tony Hong, Francis Norton, Uche Ogbuji, Glenn Olander, Mark A. Richman, Kristy Saunders, and Zoran Zaev. *Professional XML Web Services*. Wrox Press, 2001.

[9] J.D. Ullman. Information integration using logical views. In *Int. Conf. on Database Theory (ICDT)*, Delphi, Greece, 1997.

[10] Daniela Florescu, Ioana Manolescu, Donald Kossmann, and Florian Xhumari. Agora: Living with XML and Relational. In *Int. Conf. on Very Large Data Bases (VLDB)*, Cairo, Egypt, February 2000.

[11] A. Tomasic, L. Raschid, and P. Valduriez. Scaling access to heterogeneous data sources with DISCO. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):808–823, 1998.

[12] World Wide Web Consortium. XQuery 1.0: An XML Query Language. *W3C Working Draft*, December 2001.

[13] International Organization for Standardization (ISO). Information Technology-Database Language SQL. *Standard No. ISO/IEC 9075:1999*, 1999.

[14] Wolfgang Hoschek. A Database for Dynamic Distributed Content and its Application for Service and Resource Discovery. In *Int. IEEE Symposium on Parallel and Distributed Computing (submitted)*, Iasi, Romania, July 2002.

[15] Nelson Minar. Peer-to-Peer is Not Always Decentralized. In *The O'Reilly Peer-to-Peer and Web Services Conference*, Washington, D.C., November 2001.

[16] World Wide Web Consortium. Extensible Markup Language (XML) 1.0. *W3C Recommendation*, October 2000.

[17] World Wide Web Consortium. XML Schema Part 0: Primer. *W3C Recommendation*, May 2001.

[18] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. *W3C Note 15*, 2001. www.w3.org/TR/wsdl.

[19] Gnutella Community. Gnutella Protocol Specification v0.4. dss.clip2.com/GnutellaProtocol04.pdf.

[20] Ron Rivest. The MD5 message-digest algorithm. *IETF RFC 1321*, April 1992.

[21] National Institute of Standards and Technology. Secure Hash Standard. Technical report, FIPS 180-1, Washington, D.C., April 1995.

[22] Matei Ripeanu. Peer-to-Peer Architecture Case Study: Gnutella Network. In *Int. Conf. on Peer-to-Peer Computing (P2P2001)*, Linkoping, Sweden, August 2001.

[23] Clip2Report. Gnutella: To the Bandwidth Barrier and Beyond. http://www.clip2.com/gnutella.html.

[24] Donald Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, September 2000.

[25] Stefano Ceri and Giuseppe Pelagatti. *Distributed Databases - Principles and Systems.* McGraw-Hill Computer Science Series, 1985.

[26] M. Franklin, B. Jonsson, and D. Kossmann. Performance tradeoffs for client-server query processing. In *ACM SIGMOD Conf. On Management of Data*, Montreal, Canada, June 1996.

[27] Y. Papakonstantinou and V. Vassalos. Query rewriting for semistructured data. In *ACM SIGMOD Conf. On Management of Data*, 1999.

[28] Dan Suciu. Distributed Query Evaluation on Semistructured Data. *ACM Transactions on Database Systems*, 2002.

[29] T. Urhan and M. Franklin. Dynamic Pipeline Scheduling for Improving Interactive Query Performance. *The Very Large Database (VLDB) Journal*, 2001.

[30] Jordan Ritter. Why Gnutella Can't Scale. No, Really. http://www.tch.org/gnutella.html.

[31] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Workshop on Design Issues in Anonymity and Unobservability*, 2000.

[32] Apache Software Foundation. The Apache HTTP Server. http://httpd.apache.org.

[33] A. Puniyani B. Huberman L. Adamic, R. Lukose. Search in power-law networks. *Phys. Rev*, E(64), 2001.

[34] S.E. Deering. *Multicast Routing in a Datagram Internetwork.* PhD Thesis, Stanford University, 1991.

[35] J. Postel. Using SOAP in BEEP. *IETF RFC 821*, August 1982.

[36] International Telecommunications Union. Recommendation X.500, Information technology – Open System Interconnection – The directory: Overview of concepts, models, and services. *ITU-T*, November 1995.

[37] P. Mockapetris. Domain Names - Implementation and Specification. *IETF RFC 1035*, November 1987.

[38] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical report, U.C. Berkeley UCB//CSD-01-1141, 2001.

[39] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM*, 2001.

[40] M. van Steen, P. Homburg, and A. Tanenbaum. A wide-area distributed system. *IEEE Concurrency*, 1999.

[41] Steven E. Czerwinski, Ben Y. Zhao, Todd Hodes, Anthony D. Joseph, and Randy Katz. An Architecture for a Secure Service Discovery Service. In *Fifth Annual Int. Conf. on Mobile Computing and Networks (MobiCOM '99)*, Seattle, WA, August 1999.

[42] J. Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, July 1999.

[43] IEEE. *Data Engineering Bulletin*, 23(2), June 2000.

[44] Jayavel Shanmugasundaram, Kristin Tufte, David J. DeWitt, Jeffrey F. Naughton, and David Maier. Architecting a Network Query Engine for Producing Partial Results. In *WebDB 2000 (Informal Proceedings)*, 2000.

[45] Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. Integrating Network-Bound XML Data. *IEEE Data Engineering Bulletin*, 24(2), 2001.

[46] Jeffrey F. Naughton, David J. DeWitt, David Maier, Ashraf Aboulnaga, Jianjun Chen, Leonidas Galanis, Jaewoo Kang, Rajasekar Krishnamurthy, Qiong Luo, Naveen Prakash, Ravishankar Ramamurthy, Jayavel Shanmugasundaram, Feng Tian, Kristin Tufte, Stratis Viglas, Yuan Wang, Chun Zhang, Bruce Jackson, Anurag Gupta, and Rushan Chen. The Niagara Internet Query System. *IEEE Data Engineering Bulletin*, 24(2), 2001.

[47] Annita N. Wilschut and Peter M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *First Int. Conf. on Parallel and Distributed Information Systems (PDIS)*, December 1991.

[48] Zachary G. Ives, Daniela Florescu, Marc T. Friedman, Alon Y. Levy, and Daniel S. Weld. An adaptive query execution system for data integration. In *ACM SIGMOD Conf. On Management of Data*, 1999.

[49] Tolga Urhan and Michael J. Franklin. Xjoin, A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2), June 2000.

[50] Tolga Urhan and Michael J. Franklin. Dynamic Pipeline Scheduling for Improving Interactive Query Performance. In *Int. Conf. on Very Large Data Bases (VLDB)*, 2001.

[51] Beverly Yang and Hector Garcia-Molina. Efficient Search in Peer-to-Peer Networks. In *22nd Int. Conf. on Distributed Computing Systems*, Vienna, Austria, July 2002.

[52] Adriana Iamnitchi and Ian Foster. On Fully Decentralized Resource Discovery in Grid Environments. In *Int. IEEE Workshop on Grid Computing*, Denver, Colorado, November 2001.

[53] W. Yeong, T. Howes, and S. Kille. Lightweight Directory Access Protocol. *IETF RFC 1777*, March 1995.

[54] Karl Czajkowski, Steven Fitzgerald, Ian Foster, and Carl Kesselman. Grid Information Services for Distributed Resource Sharing. In *Tenth IEEE Int. Symposium on High-Performance Distributed Computing (HPDC-10)*, San Francisco, California, August 2001.

[55] Steven Fitzgerald, Ian Foster, Carl Kesselman, Gregor von Laszewski, Warren Smith, and Steven Tuecke. A Directory Service for Configuring High-Performance Distributed Computations. In *6th Int. Symposium on High Performance Distributed Computing (HPDC '97)*, 1997.